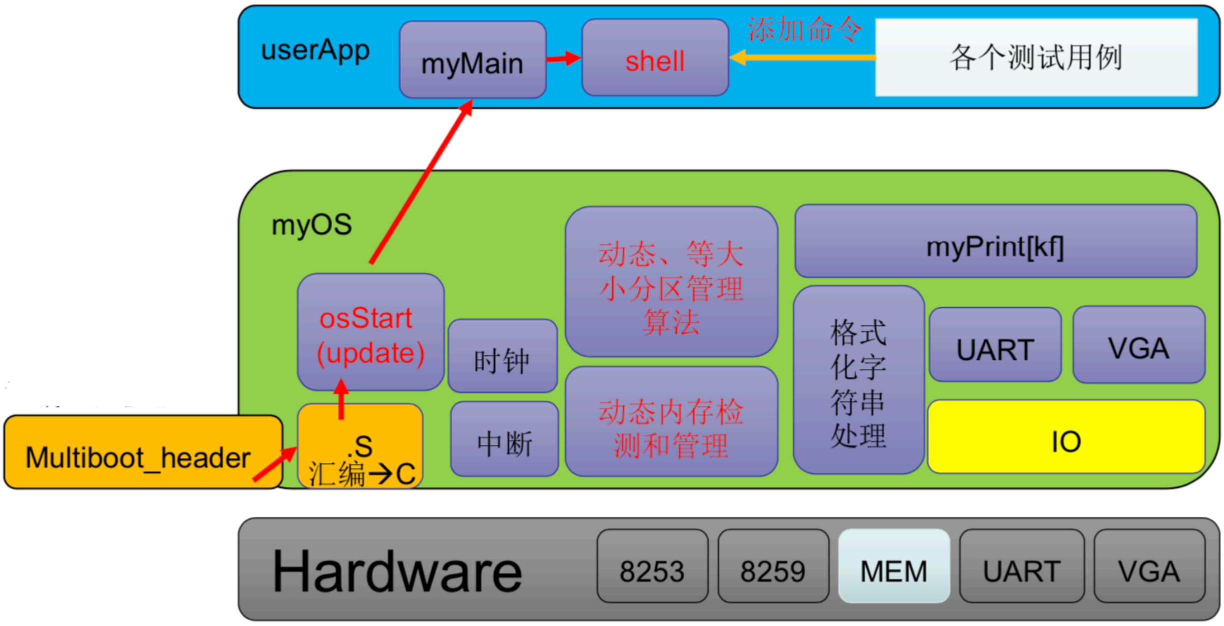
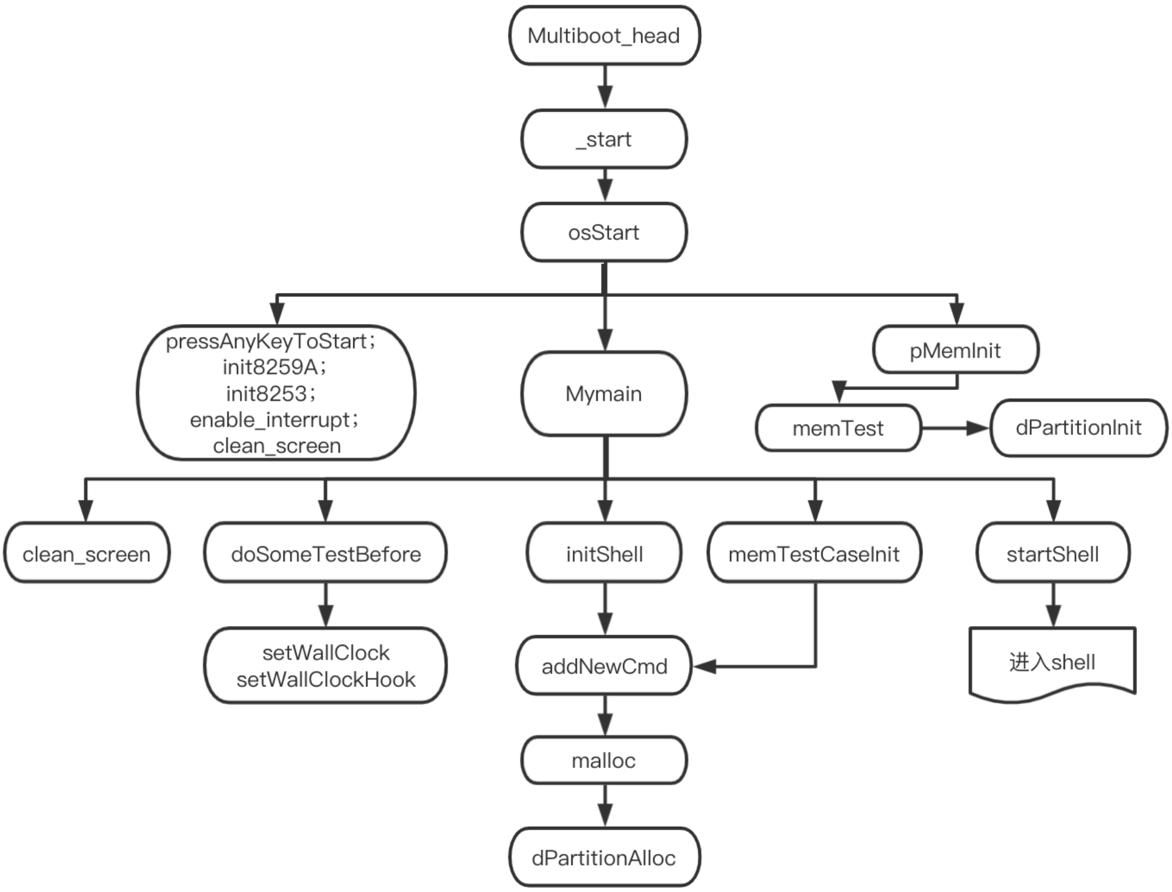
实验三 Memory Management

软件框架及概述



概述：从Multiboot_header进入入操作系统内核(myOS)，为进入C程序准备好上下文，初始化操作系统。调用myMain进入userApp。userApp中实现了用户的shell功能，shell中可调用malloc/free动态注册新命令，malloc/free由OS内核中实现的内存管理算法封装得到。

主流程及其实现



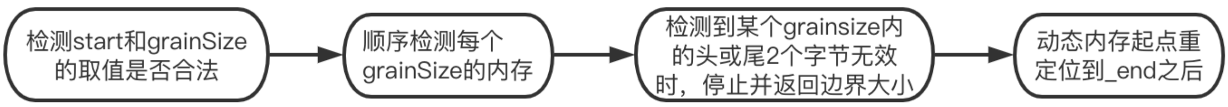
主流程图

主流程说明：multiboot_header.s调用_start入口，myOS中的Start32.s设置好中断处理，设置时钟中断(包括后续的tick维护、墙钟维护和显示)，提供了osStart()入口，做好第一次调用C语言入口前的准备，进入osStart()后，先初始化操作系统，包括初始化i8259、i8253，并开中断，检测内存并初始化，再调用与userApp之间的接口myMain，进入myMain后，清屏、打开墙钟，初始化shell并和测试用例，再调用startshell()进入自定义shell，通过malloc动态注册新命令，malloc/free根据动态分区管理算法封装得到。

主要功能模块及其实现&源代码说明

实验4的基础来自助教提供的框架

模块一 内存检测和动态内存



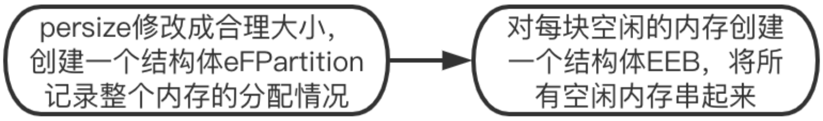

```c
void memTest(unsigned long start, unsigned long grainSize){      //内存检测
    int notFinished = 1;
    unsigned short *toTestAddr,*rearAddr;
    unsigned short temp;

    if (start<0x100000) {        //开始的地址要大于1M
```

```
        myPrintk(0x7,"????????  IN memTest: start is too small, should>=1MB  ????????\n"); while(1);
    }

    if (grainSize<0x1000) {     //默认规定grainsize要大于4kB
        myPrintk(0x7,"????????  IN memTest: grainSize is too small, should>=4KB  ????????\n"); while(1);
    }

    pMemStart = start;
    toTestAddr = (unsigned short*) start;       //unsigned short占用2个字节
    while(1){
        temp= *toTestAddr;

        *toTestAddr = 0xAA55;          //通过先覆盖再读入的方法检测grainsize的头2个字节
        if(*toTestAddr!=0xAA55) notFinished=0;

        *toTestAddr = 0x55AA;
        if(*toTestAddr!=0x55AA) notFinished=0;

        *toTestAddr = temp;             //恢复头2个字节

        rearAddr=toTestAddr+grainSize/sizeof(unsigned short)-1;
        temp= *rearAddr;

        *rearAddr = 0xAA55;           //检测grainsize的尾2个字节
        if(*rearAddr!=0xAA55) notFinished=0;

        *rearAddr = 0x55AA;
        if(*rearAddr!=0x55AA) notFinished=0;

        *rearAddr = temp;               //恢复尾2个字节

        if(notFinished) break;

        toTestAddr += grainSize/sizeof(unsigned short);     //检测下一块内存
    }

    pMemSize = (unsigned long)toTestAddr - start;

    myPrintk(0x7,"MemStart: %x  \n",pMemStart);
    myPrintk(0x7,"MemSize:  %x  \n",pMemSize);
}

extern unsigned long _end;
void pMemInit(void){                                  //初始化，包括确定动态内存
    unsigned long _end_addr = (unsigned long) &_end;
```

```
    memTest(0x100000,0x1000);
    myPrintk(0x7,"_end:  %x  \n", _end_addr);
    if (pMemStart <= _end_addr) {
        pMemSize -= _end_addr - pMemStart;         //1M到_end的部分不计入动态内存
        pMemStart = _end_addr;
    }
    pMemHandler = dPartitionInit(pMemStart,pMemSize);
}
```

模块二 等大小分区管理算法

初始化：




```
#define OVERHEAD_EFP (sizeof(struct eFPartition))
unsigned long eFPartitionInit(unsigned long start, unsigned long perSize, unsigned long n){                                  //初始化内存
    struct EEB * nextEEB;
    unsigned long nextStart = start + OVERHEAD_EFP;
    unsigned long actualSize = ((perSize + 3) >> 2) << 2;    // aligned up to 4
    struct eFPartition * theEFP = (struct eFPartition*)start;

    theEFP->totalN      = n;                         //创建一个eFPartition记录整个内存的结构
    theEFP->perSize     = actualSize;
    theEFP->firstFree = nextStart;

    for(int i=0; i<n ; i++) {                        //对每块内存创建一个EEB
        nextEEB = (struct EEB *)nextStart;
        nextStart += perSize;
        nextEEB->next_start = nextStart;
    }

    nextEEB->next_start = 0;
    return start;
}
```

计算整块内存的合理大小：

```c
unsigned long eFPartitionTotalSize(unsigned long perSize, unsigned long n){  //计算A大小
    unsigned actualSize = ((perSize + 3) >> 2) << 2;    // aligned up to 4
    return (actualSize*n+OVERHEAD_EFP);             //加上结构体eFPartition自身大小
}
```

分配内存：

```c
unsigned long eFPartitionAlloc(unsigned long EFPHandler){    //分配内存
    struct eFPartition * theEFP = (struct eFPartition*)EFPHandler;
    unsigned long addr = theEFP->firstFree;
    if (addr!=0)                        //分配第一块空闲的内存
        theEFP->firstFree = ((struct EEB*)(addr))->next_start;
    return addr;
}
```

释放内存：

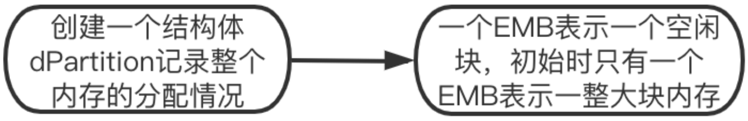
```c
unsigned long eFPartitionFree(unsigned long EFPHandler,unsigned long mbStart){ //释放内存
    struct eFPartition * theEFP;
    struct EEB* newFree;

    theEFP = (struct eFPartition*)EFPHandler;

    newFree = (struct EEB*)mbStart;
    newFree->next_start = theEFP->firstFree;  //更新EBB组成的链表
    theEFP->firstFree = mbStart;     //mbstart成为第一个空闲块

    return 1;
}
```

查看内存分配情况：

```c
void eFPartitionWalkByAddr(unsigned long efpHandler){            //查看内存分配结构
    struct eFPartition * theEFP = (struct eFPartition*)efpHandler;
    showeFPartition(theEFP);                      //打印eFPartiiton结构体的信息
    unsigned long addr = theEFP->firstFree;
    struct EEB* eeb;
    while(addr!=0){                               //遍历每一个EEB，打印出他们的地址以及下一个EEB的地址
        eeb=(struct EEB*)(addr);
        showEEB(eeb);
        addr = eeb->next_start;
    }
}
```

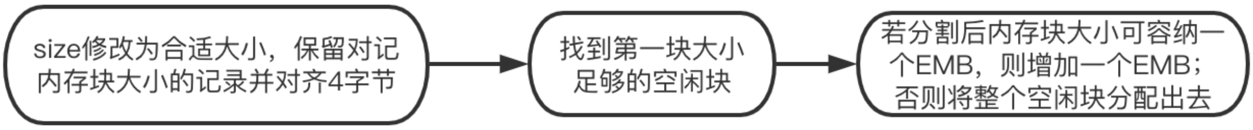
模块三 动态分区管理算法

初始化：




```c
#define MINI_DP_SIZE (sizeof(struct EMB)+sizeof(struct dPartition))
unsigned long dPartitionInit(unsigned long start, unsigned long totalSize){  //初始化内存
    struct dPartition * theDP;
    struct EMB *firstEMB;

    if (totalSize < MINI_DP_SIZE) return 0;   //totalSize至少要容纳一个EMB和一个dP

    theDP = (struct dPartition*)start;       //theDP结构体表示整个数据结构
    theDP->size  = totalSize;
    theDP->firstFreeStart = start+sizeof(struct dPartition);

    firstEMB = (struct EMB*)(theDP->firstFreeStart);       //一整块的EMB被分配,在内存中紧紧跟在dP后面
    firstEMB->size = totalSize-sizeof(struct dPartition);
    firstEMB->nextStart = 0;      //for the end of List

    return start;
}
```

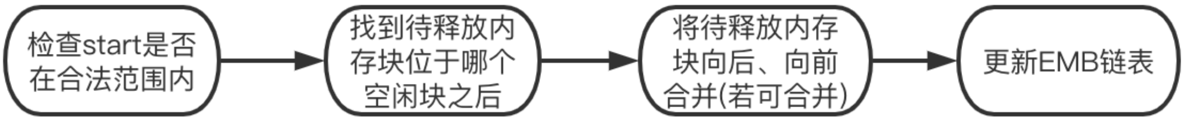
分配内存(Firstfit)：




```c
#define OVERHEAD_EMB (sizeof(struct EMB)-sizeof(struct EMB*))    //记录内存块大小所需内存
#define MINI_EMB_SIZE sizeof(struct EMB)
unsigned long dPartitionAllocFirstFit(unsigned long dp, unsigned long size){ //使用firstfit算法分配内存块
    struct dPartition *theDP;
    unsigned long curr, prev=0, next, rear;
    unsigned long actualSize = size + OVERHEAD_EMB;
    unsigned long sizeLeft;
    int notfind=1;

    actualSize = ((actualSize+3)>>2)<<2;   //align up to 4
    if (actualSize<MINI_EMB_SIZE) actualSize = MINI_EMB_SIZE;

    // find the first fit
    theDP = (struct dPartition*)dp;
    curr = theDP->firstFreeStart;

    while (notfind) {
        if (curr == 0) {
            notfind = 1;
            break;    // not find, &finish
        }

        if (actualSize<=((struct EMB*)curr)->size) {
            notfind=0;
            break;  // find, &finish
        }

        //next loop
        prev = curr;
        curr = ((struct EMB*)prev)->nextStart;
    }

    if(notfind)
        return 0;
    else {  //find
        sizeLeft = ((struct EMB*)curr)->size - actualSize;
        if (sizeLeft >= MINI_EMB_SIZE) {
            //need cut, return curr, insert rear
            rear = curr + actualSize;
            ((struct EMB*)rear)->size = ((struct EMB*)curr)->size - actualSize;
```

```
            ((struct EMB*)rear)->nextStart = ((struct EMB*)curr)->nextStart;
            if(prev) ((struct EMB*)prev)->nextStart = rear;
            else theDP->firstFreeStart = rear;

            ((struct EMB*)curr)->size =     actualSize;
        } else { // do not cut
            if(prev) ((struct EMB*)prev)->nextStart = ((struct EMB*)curr)->nextStart;
            else theDP->firstFreeStart = ((struct EMB*)curr)->nextStart;
        }

        return curr+OVERHEAD_EMB;
    }
}
```

释放内存：




```
unsigned long dPartitionFreeFirstFit(unsigned long dp, unsigned long start){  //释放内存
    unsigned long curr = start-OVERHEAD_EMB;
    unsigned long prev=0, next;
    struct dPartition *theDP;
    int notfind=1;
    theDP = (struct dPartition*)dp;
    next = theDP->firstFreeStart;
    prev = 0;

    //检查start是否在dp范围内
    if(curr<dp+sizeof(struct dPartition)||curr>=dp+theDP->size)
        return 0;

    //find position in freelist
    while(notfind){
        if ((next == 0)||(next > curr)) { // insert curr before next

            notfind = 0;
            if (prev == 0) // the first
                theDP->firstFreeStart = curr;
            else ((struct EMB*)prev)->nextStart = curr;

            ((struct EMB*)curr)->nextStart = next;

            if(next&&next==curr+((struct EMB*)curr)->size){            //向后合并
```

```
                ((struct EMB*)curr)->nextStart = ((struct EMB*)next)->nextStart;
                ((struct EMB*)curr)->size += ((struct EMB*)next)->size;
            }
            if(prev&&curr==prev+((struct EMB*)prev)->size){            //向前合并
                ((struct EMB*)prev)->nextStart = ((struct EMB*)curr)->nextStart;
                ((struct EMB*)prev)->size += ((struct EMB*)curr)->size;
            }
            break;
        }
        prev = next;
        next = ((struct EMB*)next)->nextStart;
    }
    return 1;
}
```

查看内存分配情况：

```
void dPartitionWalkByAddr(unsigned long dp){    //查看内存分配结构
    struct dPartition * theDP = (struct dPartition *) dp;
    showdPartition(theDP);                          //先印dP的信息
    unsigned long addr = theDP->firstFreeStart;
    struct EMB* emb;
    while(addr!=0){            //遍历EMB并打印其地址，大小和以及下一个EEB的地址
        emb=(struct EMB*)(addr);
        showEMB(emb);
        addr = emb->nextStart;
    }
}
```

模块四 malloc/free接口实现，shell新增命令

malloc/free接口：

```
unsigned long malloc(unsigned long size){
    dPartitionAlloc(pMemHandler,size);
}

unsigned long free(unsigned long start){
    dPartitionFree(pMemHandler,start);
}
```

Kmalloc/free接口简单实现(包含在kmalloc.h中)：

```
#define kmalloc(size)  dPartitionAlloc(pMemHandler,size)
#define kfree(start)        dPartitionFree(pMemHandler,start)
```

shell动态注册新命令：

```
void addNewCmd( unsigned char *cmd,
        int (*func)(int argc, unsigned char **argv),
        void (*help_func)(void),
        unsigned char* description){    //动态注册新命令
    struct cmd *f = (struct cmd *)malloc(sizeof(struct cmd));  //创建一个cmd的结构体
    strncpy(cmd,f->cmd,20);                         //命令名
    f->func=func;                                   //命令入口
    f->help_func=help_func;                         //命令的help入口
    strncpy(description,f->description,100);    //该命令的描述
    f->nextCmd=ourCmds;                             //维护cmd的链表
    ourCmds=f;
}
```

自测一个新命令exit，当在命令行输入exit时会退出shell。修改了startshell的部分代码帮助实现exit功能：

```
int exit_terminal(int argc, unsigned char **argv){      //exit命令入口
    if (argc>1) return 1;
    return -1;  //返回-1时，startshell接收到-1会直接return
}

void initShell(void){
    /*......其他命令......*/
    addNewCmd("exit\0",exit_terminal,NULL,"exit the terminal\0");
}

void startShell(void){
    unsigned char *argv[10];  //max 10
    int argc;
    struct cmd *tmpCmd;
    //myPrintf(0x7,"StartShell:\n");

    while(1) {
        myPrintf(0x3,"rbzhang >:");
        getCmdline(&cmdline[0],100);
        myPrintf(0x7,cmdline);

        argc = split2Words(cmdline,&argv[0],10);
        if (argc == 0) continue;

        tmpCmd = findCmd(argv[0]);
        if (tmpCmd)   {
```

```
            int rv=(tmpCmd->func(argc, argv));     //记录命令返回值
            if(rv==-1)
                return;                    //退出shell
            else if(rv==1)
                myPrintf(0x7,"UNKOWN parameter of %s\n",cmdline);  //参数不匹配
        }
        else
            myPrintf(0x7,"UNKOWN command: %s\n",argv[0]);
    }
}
```

自测kmalloc/kfree：

```
#include "../myOS/include/kmalloc.h"
#include "../myOS/include/myPrintk.h"
int test_kmalloc(int argc, unsigned char **argv){
    //======for kmalloc===============================
    char*buf1 = (char*)kmalloc(19);
    char*buf2 = (char*)kmalloc(24);

    for(int i=0;i<17;i++) *(buf1+i) = '*';
    *(buf1+17) = '\n';
    *(buf1+18) = '\000';

    for(int i=0;i<22;i++) *(buf2+i) = '#';
    *(buf2+22) = '\n';
    *(buf2+23) = '\000';

    myPrintk(0x5, "We allocated 2 buffers.\n");
    myPrintk(0x5, "BUF1(size=19, addr=0x%x) filled with 17(*): ",(unsigned long)buf1);
    myPrintk(0x7,buf1);
    myPrintk(0x5, "BUF2(size=24, addr=0x%x) filled with 22(#): ",(unsigned long)buf2);
    myPrintk(0x7,buf2);

    myPrintk(0x7,"\n");

    kfree((unsigned long)buf1);
    kfree((unsigned long)buf2);

    return 0;
}
void memTestCaseInit(void){
    /*......其他命令......*/
    addNewCmd("testKmalloc\0", test_kmalloc, NULL, "Kmalloc, write and read.\0");
}
```

目录组织

```
.
├── Makefile
├── README_mem.txt
├── multibootheader
│   └── multibootHeader.S
├── myOS
│   ├── Makefile
│   ├── dev
│   │   ├── Makefile
│   │   ├── i8253.c
│   │   ├── i8259A.c
│   │   ├── uart.c
│   │   └── vga.c
│   ├── i386
│   │   ├── Makefile
│   │   ├── io.c
│   │   ├── irq.S
│   │   └── irqs.c
│   ├── include
│   │   ├── i8253.h
│   │   ├── i8259.h
│   │   ├── io.h
│   │   ├── irq.h
│   │   ├── kmalloc.h
│   │   ├── malloc.h
│   │   ├── mem.h
│   │   ├── myPrintk.h
│   │   ├── string.h
│   │   ├── uart.h
│   │   ├── vga.h
│   │   ├── vsprintf.h
│   │   └── wallClock.h
│   ├── kernel
│   │   ├── Makefile
│   │   ├── mem
│   │   │   ├── Makefile
│   │   │   ├── dPartition.c
│   │   │   ├── eFPartition.c
│   │   │   ├── malloc.c
│   │   │   └── pMemInit.c
│   │   ├── tick.c
│   │   └── wallClock.c
│   ├── lib
│   │   ├── Makefile
│   │   └── string.c
│   ├── myOS.ld
│   ├── osStart.c
│   ├── printk
│   │   ├── Makefile
│   │   ├── myPrintk.c
│   │   ├── types.h
│   │   └── vsprintf.c
│   ├── start32.S
│   └── userInterface.h
├── source2img.sh
└── userApp
    ├── Makefile
    ├── main.c
    ├── memTestCase.c
    ├── memTestCase.h
    ├── shell.c
    └── shell.h
```

Makefile组织

关键规则：

```
output/myOS.elf: ${OS_OBJS} ${MULTI_BOOT_HEADER}
	${CROSS_COMPILE}ld -n -T myOS/myOS.ld ${MULTI_BOOT_HEADER} ${OS_OBJS} -o output/myOS.elf

output/%.o : %.S	#所有的.s生成.o
	@mkdir -p $(dir $@)
	@${CROSS_COMPILE}gcc ${ASM_FLAGS} -c -o $@ $<

output/%.o : %.c	#所有的.c生成.o
	@mkdir -p $(dir $@)
	@${CROSS_COMPILE}gcc ${C_FLAGS} -c -o $@ $<
```

先由各级子目录下的.c 和.s文件生成.o文件，再将.o文件作为依赖文件，按照myOS.ld规则链接成终极目标文件myOS.elf。

代码布局说明

首先定位到内存中1M地址处。可执行文件的.text段从此处开始。先存放.multiboot_header段[12字节]，往后对齐8字节后，再存放所有输入文件的.text段。 往后对齐16字节，开始存放可执行文件的.data段，即为所有输入文件的.data段。往后对齐16字节，接着存放可执行文件的.bss段，包括所有输入文件中未初始化的全局变量。bss段结束后再向后对齐16字节，此处以_end作为结束标记。往后对齐512字节。

编译过程说明

直接在终端运行./source2run.sh即可。具体过程是先按照makefile内容进行编译链接，编译成功后再执行命令

```
qemu-system-i386 -kernel output/myOS.elf -serial pty &
```

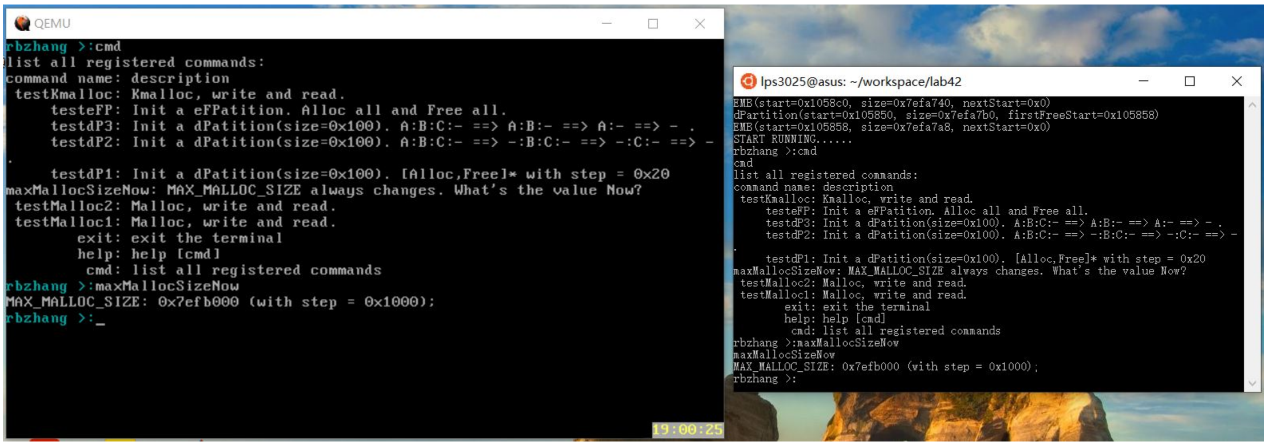
将串口重定向到伪终端，运行时会告知具体是哪个，据此输入

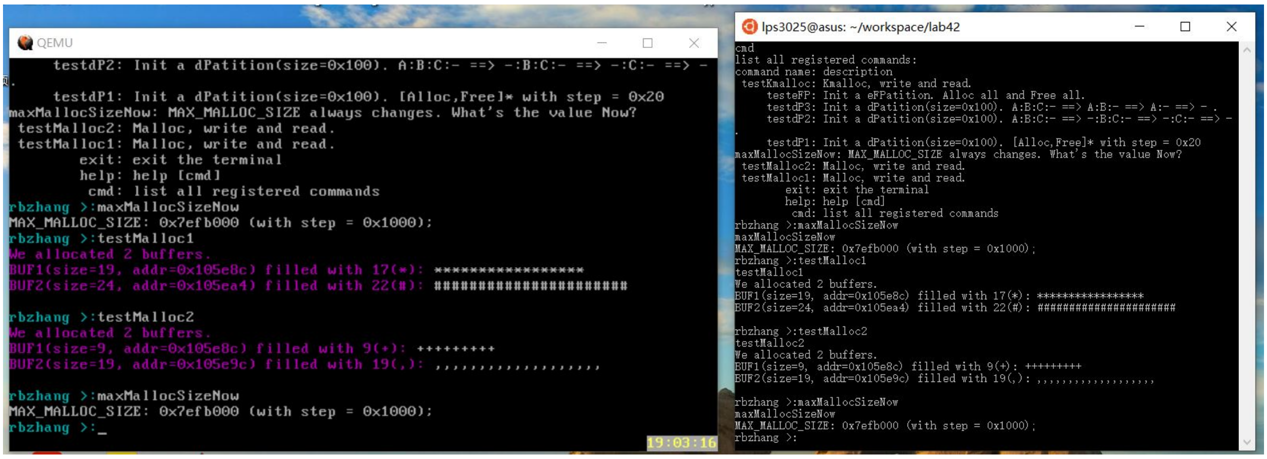
```
sudo screen /dev/pts/0   #假设是/dev/pts/0
```

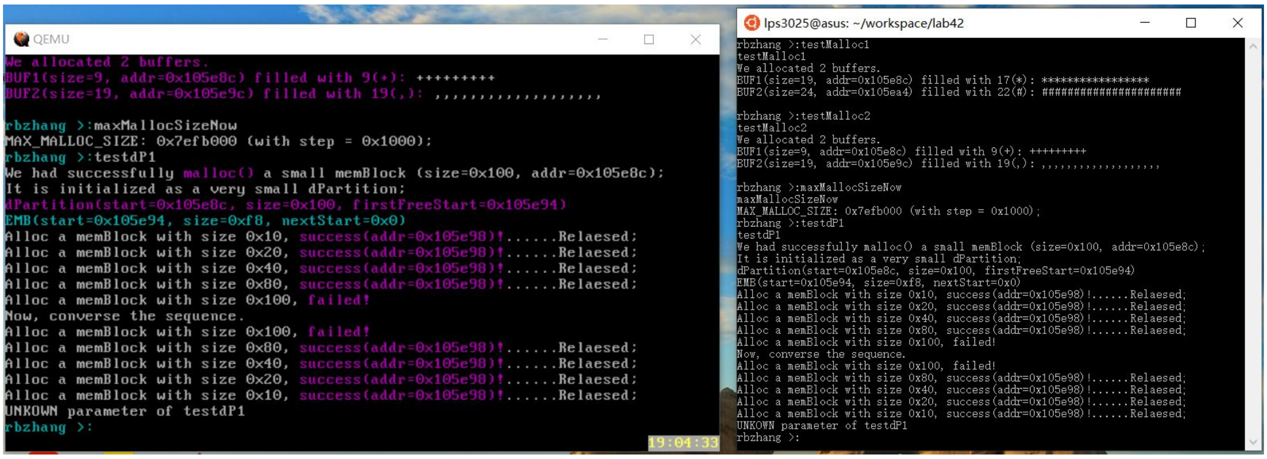
接着就可以通过伪终端输入命令。

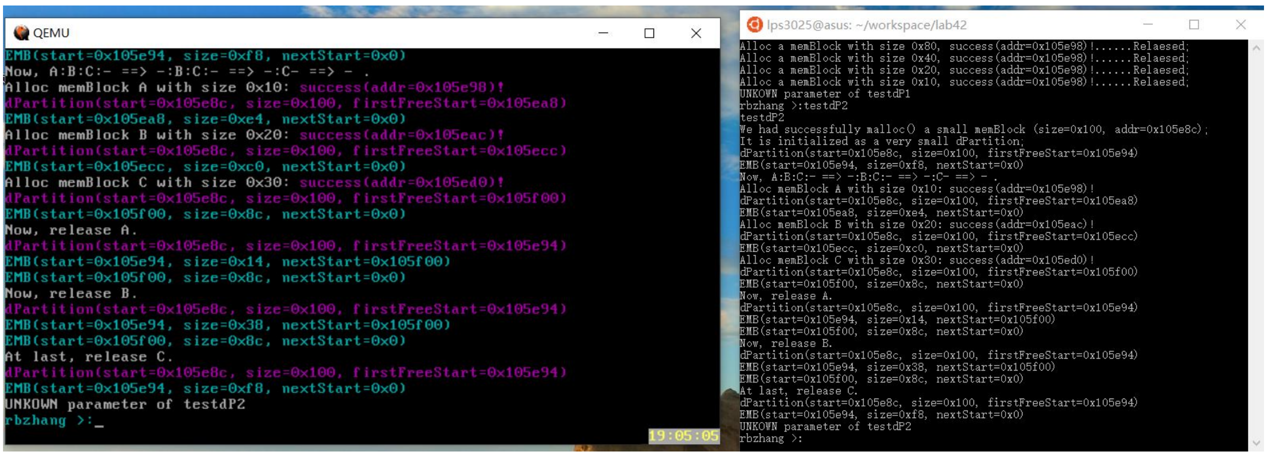
运行和运行结果说明

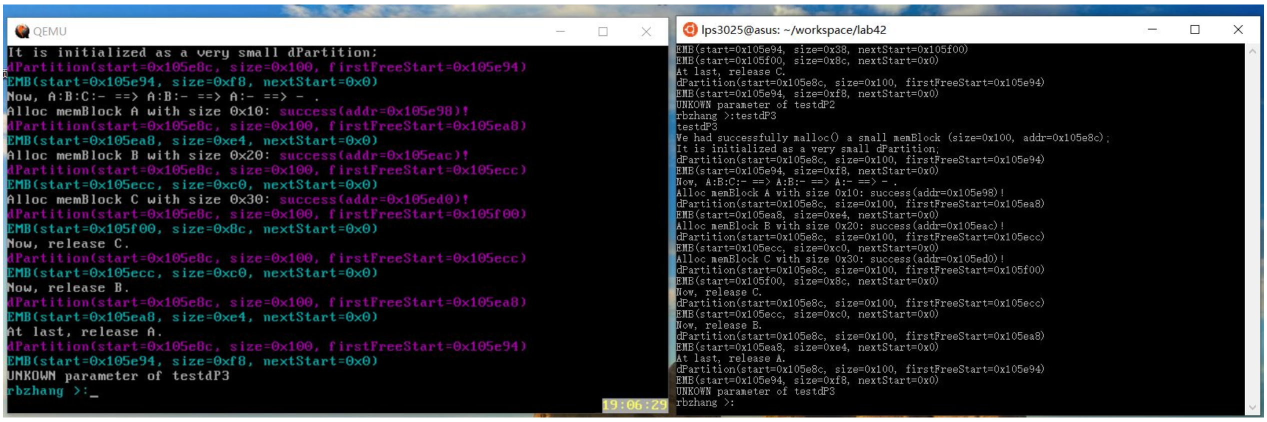
运行结果如下：

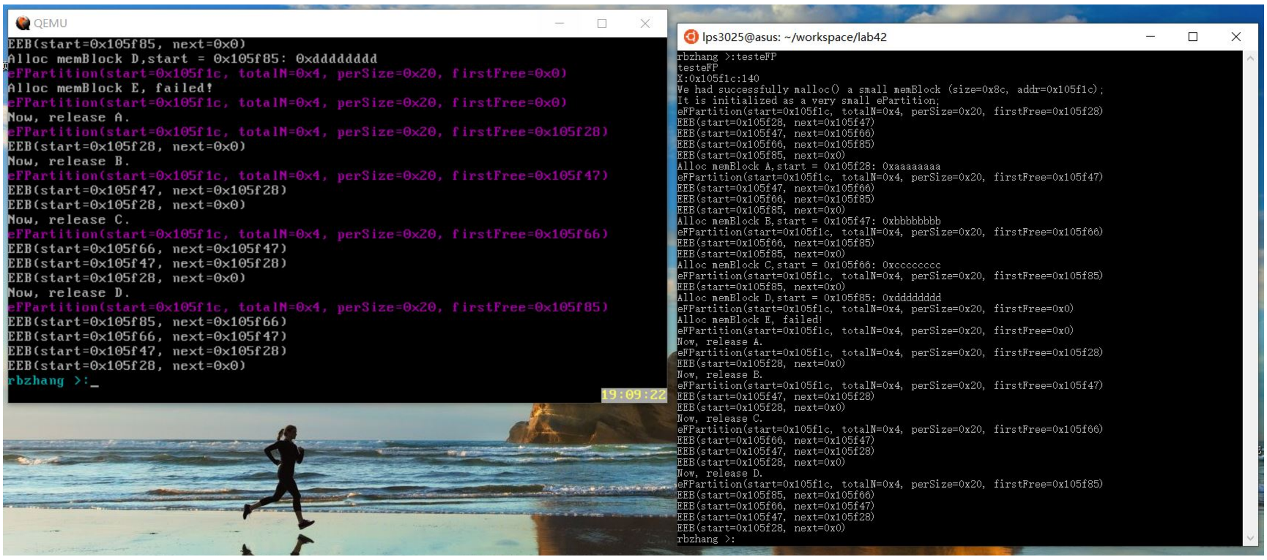

QEMU
rbzhang >:cmd
list all registered commands:
command name: description
testKmalloc: Kmalloc, write and read.
testeFP: Init a eFPatition. Alloc all and Free all.
testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
exit: exit the terminal
help: help [cmd]
cmd: list all registered commands
rbzhang >:maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:_
19:00:25
lps3025@asus: ~/workspace/lab42
EMB(start=0x1058c0, size=0x7efa740, nextStart=0x0)
dPartition(start=0x105850, size=0x7efa7b0, firstFreeStart=0x105858)
EMB(start=0x105858, size=0x7efa7a8, nextStart=0x0)
START RUNNING......
rbzhang >:cmd
cmd
list all registered commands:
command name: description
testKmalloc: Kmalloc, write and read.
testeFP: Init a eFPatition. Alloc all and Free all.
testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
exit: exit the terminal
help: help [cmd]
cmd: list all registered commands
rbzhang >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:

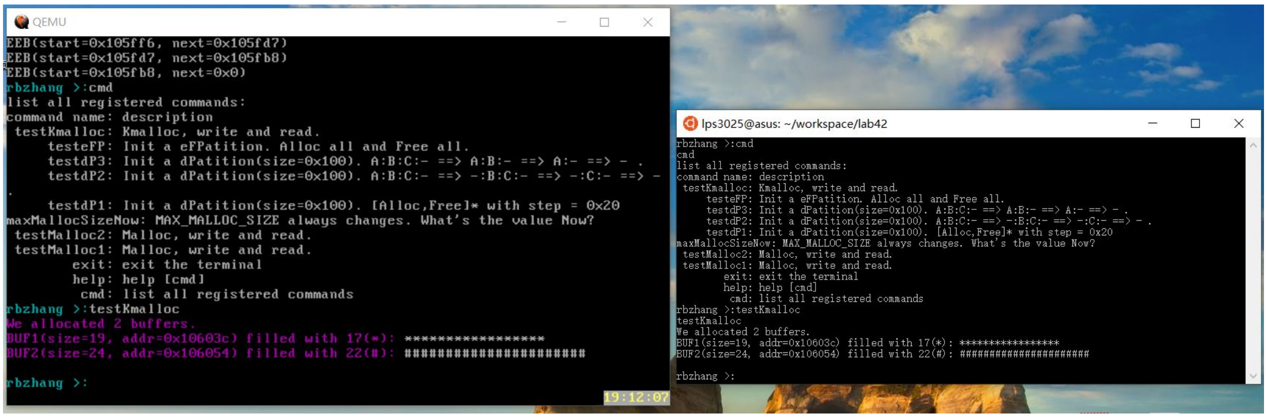

QEMU
testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
exit: exit the terminal
help: help [cmd]
cmd: list all registered commands
rbzhang >:maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105e8c) filled with 17(*): *****************
BUF2(size=24, addr=0x105ea4) filled with 22(#): ######################
rbzhang >:testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x105e8c) filled with 9(+): +++++++++
BUF2(size=19, addr=0x105e9c) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
rbzhang >:maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:_
19:03:16
lps3025@asus: ~/workspace/lab42
cmd
list all registered commands:
command name: description
testKmalloc: Kmalloc, write and read.
testeFP: Init a eFPatition. Alloc all and Free all.
testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> -
testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
exit: exit the terminal
help: help [cmd]
cmd: list all registered commands
rbzhang >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:testMalloc1
testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105e8c) filled with 17(*): *****************
BUF2(size=24, addr=0x105ea4) filled with 22(#): ######################
rbzhang >:testMalloc2
testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x105e8c) filled with 9(+): +++++++++
BUF2(size=19, addr=0x105e9c) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
rbzhang >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:


QEMU
We allocated 2 buffers.
BUF1(size=9, addr=0x105e8c) filled with 9(+): +++++++++
BUF2(size=19, addr=0x105e9c) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
rbzhang >:maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:testdP1
We had successfully malloc() a small memBlock (size=0x100, addr=0x105e8c);
It is initialized as a very small dPartition;
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0xf8, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x20, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x40, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x80, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x100, failed!
Now, converse the sequence.
Alloc a memBlock with size 0x100, failed!
Alloc a memBlock with size 0x80, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x40, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x20, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x10, success(addr=0x105e98)!......Released;
UNKOWN parameter of testdP1
rbzhang >:
19:04:33
lps3025@asus: ~/workspace/lab42
rbzhang >:testMalloc1
testMalloc1
We allocated 2 buffers.
BUF1(size=19, addr=0x105e8c) filled with 17(*): *****************
BUF2(size=24, addr=0x105ea4) filled with 22(#): ######################
rbzhang >:testMalloc2
testMalloc2
We allocated 2 buffers.
BUF1(size=9, addr=0x105e8c) filled with 9(+): +++++++++
BUF2(size=19, addr=0x105e9c) filled with 19(,): ,,,,,,,,,,,,,,,,,,,
rbzhang >:maxMallocSizeNow
maxMallocSizeNow
MAX_MALLOC_SIZE: 0x7efb000 (with step = 0x1000);
rbzhang >:testdP1
testdP1
We had successfully malloc() a small memBlock (size=0x100, addr=0x105e8c);
It is initialized as a very small dPartition;
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0xf8, nextStart=0x0)
Alloc a memBlock with size 0x10, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x20, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x40, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x80, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x100, failed!
Now, converse the sequence.
Alloc a memBlock with size 0x100, failed!
Alloc a memBlock with size 0x80, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x40, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x20, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x10, success(addr=0x105e98)!......Released;
UNKOWN parameter of testdP1
rbzhang >:


QEMU
EMB(start=0x105e94, size=0xf8, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105e98)!
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105ea8)
EMB(start=0x105ea8, size=0xe4, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105eac)!
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105ecc)
EMB(start=0x105ecc, size=0xc0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105ed0)!
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105f00)
EMB(start=0x105f00, size=0x8c, nextStart=0x0)
Now, release A.
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0x14, nextStart=0x105f00)
EMB(start=0x105f00, size=0x8c, nextStart=0x0)
Now, release B.
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0x38, nextStart=0x105f00)
EMB(start=0x105f00, size=0x8c, nextStart=0x0)
At last, release C.
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0xf8, nextStart=0x0)
UNKOWN parameter of testdP2
rbzhang >:_
19:05:05
lps3025@asus: ~/workspace/lab42
Alloc a memBlock with size 0x80, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x40, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x20, success(addr=0x105e98)!......Released;
Alloc a memBlock with size 0x10, success(addr=0x105e98)!......Released;
UNKOWN parameter of testdP1
rbzhang >:testdP2
testdP2
We had successfully malloc() a small memBlock (size=0x100, addr=0x105e8c);
It is initialized as a very small dPartition;
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0xf8, nextStart=0x0)
Now, A:B:C:- ==> -:B:C:- ==> -:C- ==> - .
Alloc memBlock A with size 0x10: success(addr=0x105e98)!
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105ea8)
EMB(start=0x105ea8, size=0xe4, nextStart=0x0)
Alloc memBlock B with size 0x20: success(addr=0x105eac)!
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105ecc)
EMB(start=0x105ecc, size=0xc0, nextStart=0x0)
Alloc memBlock C with size 0x30: success(addr=0x105ed0)!
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105f00)
EMB(start=0x105f00, size=0x8c, nextStart=0x0)
Now, release A.
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0x14, nextStart=0x105f00)
EMB(start=0x105f00, size=0x8c, nextStart=0x0)
Now, release B.
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0x38, nextStart=0x105f00)
EMB(start=0x105f00, size=0x8c, nextStart=0x0)
At last, release C.
dPartition(start=0x105e8c, size=0x100, firstFreeStart=0x105e94)
EMB(start=0x105e94, size=0xf8, nextStart=0x0)
UNKOWN parameter of testdP2
rbzhang >:







说明：分别在命令行输入cmd, maxMallocSizeNow, testMalloc1, testMalloc2, maxMallocSizeNow, testdP1,t estdP2, testdP3, testeFP，最后输入kmalloc/free的自编测试命令testKmalloc。

遇到的问题和解决方案说明

使用动态分区管理算法释放内存时，需要知道被分配的内存大小。因此需要在分配空闲块的时候，就将该处EMB的size信息保留下来，即从内存中size存储位置的后面开始分配所需大小的内存。